Universidade Federal de Campina Grande

Guilherme de Melo Carneiro	118210938
Jonathan Tavares da Silva	118210631
Mirella Quintans Lyra	118210425
Túlio Araújo Cunha	118210965

Relatório de Projeto LP2 2019.1

Professor Orientador: Eliane Araújo
Monitor orientador: Lucas de Medeiros Nunes Fernandes

Campina Grande - PB
28 de maio de 2019

Ferramentas

- Diagrama de classes: Draw.io, IntelliJ IDEA
- Versionamento de código: Github
- Comunicação: Slack, WhatsApp, Jitsi
- Notificações do Github no Slack: Github Bot

Design geral

Para fornecer uma solução para a especificação proposta, fizemos uso contínuo do padrão *Controller*, que agiu como *Creator* na maioria das situações, o que permitiu uma maior coesão das classes que representavam entidades, como *Projeto* e *Pessoa*. Também usou-se interfaces para modelar o processamento de entidades com comportamentos diferentes, mas com proximidade de características - como foi o caso da interface *PropostaLegislativa* para generalizar *PEC*, *PLP* e *PL* -, o que permitiu um menor acoplamento e também uma maior coesão, além de facilitar possíveis implementações futuras de novos tipos de propostas. Também foi utilizado o padrão *Strategy* para se adequar a comportamentos mutáveis, como foi o caso da busca por proposta mais relacionada, onde foram criados vários *Comparators* específicos, utilizados de acordo com a configuração do usuário. *Enumerators* foram utilizados para comportamentos imutáveis, como foi o caso dos tipos de proposta *PEC*, *PLP* e *PEC*, além de método de validação de entrada, como no caso da configuração da estratégia de busca de proposta mais relacionada, *APROVACAO*, *CONCLUSAO* e *CONSTITUCIONAL*. Também implementou-se uma classe validadora, responsável somente pela validação das entradas, lançando exceções para entradas inválidas, e também uma classe buscadora, responsável pelos algoritmos de busca de propostas cadastradas. A organização dos pacotes se deu por semelhança de funções, isolando-se *controllers*, entidades, *enumerators*, classes utilitárias, entre outras.

Caso 1 - Cadastro de Pessoas

O caso 1 pede que sejam implementados métodos para cadastrar uma pessoa no E-Camara Organizada. Uma pessoa pode ser cadastrada de duas formas distintas, sendo elas: com partido ou sem partido. A implementação do cadastro de pessoa se deu com a criação de uma classe *PessoaController* que possui um atributo *pessoas* no qual serão armazenadas as pessoas cadastradas. O atributo *pessoas* é um mapa que tem como chave o documento nacional de identificação da pessoa e aponta para um objeto do tipo *Pessoa*. *Pessoa* é uma classe-entidade que foi criada para representar uma Pessoa no sistema. Pessoa tem documento de identificação, nome, estado, interesses e partido (estes dois últimos são opcionais). O cadastro de Pessoa foi implementado com a sobrecarga do método *cadastrarPessoa()*, a primeira implementação com o parâmetro partido e a segunda sem o parâmetro. Implementou-se métodos para validar as informações (parâmetros) recebidos e lançar possíveis exceções como *NullPointerException* e *IllegalArgumentException*. Por fim, utilizou-se expressão regular para validar o documento nacional de identificação da pessoa no cadastro, que tem o formato “000000000-0”.

Caso 2 - Cadastro de Deputados

No caso de uso 2 é pedido que seja feita a implementação do cadastro de Deputados. Para resolver o problema, decidiu-se criar uma interface *CargoPolitico* que define um tipo para todos os cargos políticos que uma pessoa possa vir a assumir, facilitando assim uma possível troca. Criada a interface, a classe *Pessoa* agora tem uma composição de *CargoPolitico*. Falando agora especificamente sobre o Deputado, criou-se uma classe *Deputado* para representar o cargo específico no sistema, possuindo o atributo leis e o atributo dataInicio, que representam o número de leis aprovadas na Câmara e a data de início do mandato, respectivamente. Além disso, possui a implementação dos métodos da interface *getNomeCargo()*, *getDataInicio()* e *getLeis()*. A classe implementa a interface *CargoPolitico*, sendo assim, uma pessoa pode ser um Deputado. Só é possível cadastrar um Deputado se existir uma pessoa que possua o documento nacional recebido como parâmetro e devidamente validado, essa mesma pessoa também deve possuir um partido. Por fim, a data de início foi validada utilizando as classes nativas *Date*, *DateFormat* e *SimpleDateFormat*.

Caso 3 - Exibição de Pessoas/Deputados

O caso 3 pede que seja implementada a função de exibição de pessoas/deputados, levando em consideração o fato de que a pessoa que tem um cargo político vai ter uma representação diferente de uma pessoa que não tem cargo, e que quando é somente a exibição de uma pessoa sem cargo, ela pode ter ou não partido e interesses, assim como o deputado pode ter ou não interesses. Para resolver o problema, foi feita a implementação de *exibirPessoa* no *PessoaController* que recebe um DNI (documento nacional de identificação) que vai ser validado antes de tentar consultar aquela pessoa/deputado na estrutura do mapa de pessoas. Ao ser invocado, o método acessa o *toString()* da classe *Pessoa*, que possui todas as validações e testes adequados para formar a *string* que vai ser retornada. Foi criado um método privado chamado de *informacoesBasicas()* que retorna a formação de uma *string* que é comum tanto a deputado como a pessoa sem cargo, sendo assim, o código é reaproveitado com base no padrão *DRY*. No método *toString()* de *Pessoa* mais especificamente, é validado se interesses é vazio e se partido é vazio, com isso feito, é formada a *string* que representa uma pessoa para exibição.

Caso 4 - Cadastro de Partidos Governistas

Nesse caso, é pedido que seja implementado o cadastro e exibição de partidos que compõem a base governista, com a exibição em ordem lexicográfica. Para isso, foi criado o controller de partidos que se encarrega de guardar todos os partidos da base governista e operar sobre eles. Criou-se também a entidade *Partido*, que possui como único atributo seu nome, e foi feita a implementação do *cadastrarPartido()* em *PartidoBaseController*, que recebe o nome do partido que será validado para valores nulos ou vazios antes de ser, de fato, cadastrado no sistema. O sistema não permite o cadastro de um partido mais de uma vez. Foi criado um mapa dentro de *PartidoController* que gere os Partidos cadastrados utilizando como chave seus nomes. Para exibir os Partidos de base em ordem lexicográfica foi utilizada a classe ComparatorOrdemAlfabeticaPartido como Comparator para organizar a saída do método *exibirBase()*.

Caso 5 - Cadastrar Comissão

O caso 5 pede que seja implementado o cadastro de comissões com base em seu tema e seus integrantes. Para isso, foi criada a entidade Comissão, que é identificada unicamente pelo seu tema, e que é integrada por Pessoas cadastradas através dos seus respectivos DNIs. Também foi criado uma classe *ComissaoController*, responsável por criar e gerir Comissões usando padrão *Creator*. Foi utilizado um *HashMap* dentro de *ComissaoController* com o tema de cada comissão como chave de acesso. O cadastro de uma Comissão foi feito a partir de um método *cadastraComissao()*, que cria e salva uma Comissão com base em seu tema e integrantes. O sistema não permite a criação de Comissões sem integrantes ou de Comissões que já tenham sido cadastradas. Entradas nulas ou inválidas lançam *NullPointerException* e *IllegalArgumentException*, respectivamente. Também foi utilizado na classe *ComissaoController* a classe *PessoaService*, para que haja a validação do cadastro somente de deputados/pessoas que já pertençam ao sistema.

Caso 6 - Cadastrar e Exibir Proposta Legislativa

Essa etapa do projeto tem como finalidade permitir o cadastro e exibição de propostas legislativas. Tais propostas podem ser do tipo PL, PLP ou PEC. Devido à semelhança tanto entre os atributos destes três tipos de propostas legislativas como entre seus métodos, optou-se por trabalhar com composição. Desta forma criou-se uma interface *PropostaLegislativa* que é implementada pela classe abstrata Projeto e, as classes *PL*, *PLP* e *PEC* são extensões da classe *Projeto*.
Usou-se um *enum TipoProjeto* para armazenar as constantes possíveis para este atributo.
A classe *ProjetoController* foi criada com o intuito de gerir as novas funcionalidades que foram implantadas no projeto nesta etapa. Ela contém um *HashMap* de objetos do tipo *Projeto* cuja chave é um código que associa o tipo do projeto ao seu ano e ordem de criação respectivamente.
Antes de cadastrar um projeto todas as entradas recebidas são validadas. Além da verificação de *strings* nulas ou vazias, é verificado também se a *DNI* do autor do projeto é válida e pertence a uma pessoa que possui cargo político, para tanto utiliza-se a classe *PessoaService*. Além disso, verifica-se se o ano informado é um ano válido, ou seja, posterior a 1988 e menor ou igual ao ano corrente. Na constatação de entradas inválidas exceções do tipo *NullPointerException* e *IllegalArgumentException* sao lançadas. A exibição das propostas é feita através da inserção do código da proposta buscada pelo usuário, caso o código seja válido a proposta é exibida, caso contrário uma exceção é lançada.

Caso 7 - Votar Proposta Legislativa

Nesta etapa do projeto o objetivo é votar uma Proposta Legislativa. Nesse processo uma proposta pode assumir três estados: em votação, aprovada ou rejeitada, novamente utilizou-se um *enum* armazenando esses estados como constantes.
As votações podem ocorrem nas comissões ou no plenário. Para votações ocorridas no plenário as regras para aprovação ou rejeição são as mesmas para todos os tipos de Proposta Legislativa, por isso, o método *votarComissao()* foi implementado na classe abstrata *Projeto*.
As votações no plenário, por sua vez, possuem particularidades no seu processo de aprovação de acordo com o tipo do projeto a que dizem respeito. Em função disso, optou-se por trabalhar utilizando o padrão *Strategy*, assim o método *votarPlenario()* foi criado dentro da classe abstrata

Projeto e cada uma das classes *PL*, *PLP* e *PEC* teve sua implementação feita internamente respeitando suas especificidades.
Para ambos os métodos de votação verifica-se a presença de entradas nulas ou vazias e, caso encontradas, exceçoes do tipo *NullPointerException* ou *IllegalArgumentException* são lançadas. Para votação em plenário verifica-se também se o quorum mínimo é atingido. Caso não o seja, uma exceção é lançada.

Caso 8 - Exibir Tramitação de PL

O caso 8 adiciona ao sistema a possibilidade de exibição da tramitação de uma PL. Para implementar essa funcionalidade, criou-se dentro da classe *Projeto* o atributo votações que é uma lista de *string* que armazena o status da votação associado ao local onde o *Projeto* foi votado. O método *exibirTramitacao()* itera sobre essa coleção para retornar a *string* com a informação desejada para o usuário.
Por meio do *ProjetoController* se acessa o *HashMap* de projetos para exibir a tramitação da PL desejada inserindo o código do mesmo como chave de acesso. Este código de acesso é validado verificando se, de fato, existe uma PL cadastrada no sistema com aquele código. Em caso de inconsistência uma exceção do tipo *NullPointerException* ou *IllegalArgumentException* é lançada.

Caso 9 - Pegar Proposta Mais Relacionada

Nesse caso, retorna-se a proposta mais relacionada a uma pessoa cadastrada de acordo com seus interesses, adotando critérios de desempate alternativos caso se faça necessário, de acordo com a configuração definida pelo usuário, podendo ser APROVACAO, CONSTITUCIONAL e CONCLUSAO.
A estratégia utilizada foi o padrão *Strategy* para a definição do critério de desempate, sendo criado um Comparator específico para cada tipo(APROVACAO, CONSTITUCIONAL e CONCLUSAO), além dos tipos genéricos(menor ano, primeiro cadastro), permitindo assim a criação de novos critérios de desempate sem muito esforço, se necessário. Para a execução da busca, foi criada uma classe *Buscador*, no pacote util, responsável por pesquisar a proposta mais relacionada de acordo com a prioridade definida na especificação. A classe Buscador pode adotar métodos que buscam com base nos interesses e nos critérios desejados, se tornando expansível. Fez-se uma composição na classe *ProjetoController* com um Buscador.
Para filtrar as propostas mais relacionadas a uma pessoa, utilizou-se o artifício do Java 8 de programação funcional *Stream*, permitindo excluir da busca propostas fora dos critérios. Também utilizou-se classes Comparators para cada atributo utilizado no algoritmo de busca, para que prevalecesse sempre a ordem de desempate: “Maior Nº de interesses em comum > Critério de desempate(APROVACAO, CONSTITUCIONAL ou CONCLUSAO) > Ano de criação > ordem de cadastro.
Para guardar a informação de qual foi a ordem de cadastro das propostas legislativas, criou-se um contador ***static*** na classe Projeto, que atribui a cada objeto dos tipos que herdam de Projeto um valor único e sequencial de acordo com o número salvo no contador ***static*** da classe Projeto.

Caso 10 - Persistir Dados

Para resolver o caso 10, ou seja, fazer com que os dados fossem assegurados mesmo com o desligamento do sistema, ou a ocorrência de algum outro problema, criou-se uma classe controladora chamada *PersistenciaController* que vai possuir os três métodos básicos públicos e

acessíveis à Facade. Criou-se também uma classe de Service para comunicar o *PersistenciaController* com o *ProjetoController* e por consequência, como *ProjetoController* possui instância dos outros *services* de comunicação, ter acesso a eles pelo *ProjetoService*. Tendo Comunicado todos os controllers, agora o *PersistenciaController* tem acesso a todos os mapas que guardam os objetos de entidade do sistema. Por meio das classes de Service de cada Controller, criou-se métodos get e set que irão servir para retornar a estrutura no momento de salvar no arquivo e setar a estrutura quando o método *carregarSistema()* é chamado. Em PersistenciaController criou-se métodos privados auxiliares (Salvar, Recuperar e Carregar) para deixar claro cada etapa de persistência dos arquivos e de leitura dos objetos. Cada estrutura de mapa é salva em um arquivo diferente com seu respectivo nome. Por fim, para a persistência, utilizou-se as classes de arquivos e fluxo de dados *File*, *FileInputStream*, *FileOutputStream*, *FileWriter*, *ObjectInputStream* e *ObjectOutputStream*.

Considerações

A construção do E-Camara Organizada nos trouxe uma perspectiva real das dificuldades e desafios enfrentados por uma equipe de desenvolvedores no momento de criação de um programa a partir das especificações do cliente. Neste tipo de situação, cabe aos desenvolvedores não só elaborar um código que atenda às necessidades atuais do cliente, como também que envolva todos os pilares da programação OO aprendidos em sala de aula. Por isso, nossas escolhas de design foram sempre com intuito de ter um código expansível, manutenível e reutilizável.

Um importante ponto deste trabalho é a real compreensão do significado de trabalho em equipe, situação praticamente mandatória nas empresas de tecnologia atuais. Assim, a preocupação com o desenvolvimento de um código limpo foi constante na nossa equipe ao longo de todas as US's.

A partir do E-Camara Organizada passamos a ter uma melhor compreensão sobre o sistema de aprovação de leis vigente em nosso país, algo que devia ser imperativo para qualquer cidadão brasileiro.

De forma geral, este trabalho nos acrescentou conhecimentos não só quanto à design de um sistema, mas nos permitiu aprofundar os conhecimentos em Java, a partir das discussões de novas funcionalidades antes desconhecidas para alguns ou todos os membros da equipe, desenvolver nossas habilidades de relação interpessoais, lidar com o conceito de prazo, aperfeiçoar os conhecimentos da plataforma git, além de nos fazer compreender os mecanismos de votação e aprovação de alguns projetos de lei do nosso sistema político.

Link para o repositório no GitHub

https://github.com/jonathantvrs/e-co.git